Science of Computer Programming 76 (2011) 278–289

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Automating exception-safety classification

Gustav Munkby *, Sibylle Schupp

Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden

ARTICLE INFO

Article history:
Received 23 February 2007
Received in revised form 20 May 2008
Accepted 4 June 2008
Available online 29 June 2008

Keywords:
Strong exception-safety guarantee
Rollback semantics
Data-flow analysis
Program safety
Exception handling

ABSTRACT

Exception handling mechanisms provide a structured way to deal with exceptional circumstances, making it easier to read and reason about programs. Exception handling, however, cannot avoid the problem that the transfer of control might leave the program in an inconsistent state—resources might leak, invariants might be violated, the program state might be changed. Since client code often needs to know how a program behaves in the presence of exceptions, the *exception-safety classification* distinguishes three different classes of safety guarantees; this classification is used, for example, during the review process in the Boost organization for standardized libraries in C++. Classifying the safety level of a procedure requires understanding program invariants and tracking program state at any given point in the code, which is error-prone when done by hand. Yet, no tool support is available to date. In this paper we present the first automated analysis for exception guarantees. Since the safety level of an arbitrary procedure is undecidable, the analysis conservatively approximates exception safety . The analysis is based on the theory of backward data-flow analysis and recognizes two of the three safety guarantees, the *strong* and the *no-throw* guarantee, and provides counterexamples otherwise. A prototype implementation is available.

1. Introduction

In libraries and other modern software systems it is common to employ exception-handling mechanisms to deal with exceptional situations. Instead of intertwining normal code and error-handling code, guarding a possibly great number of single expressions with error checks, and introducing multiple exit points of a procedure, exception handling allows separating all error-handling code from the normal code and encapsulating it in exception handlers. This separation makes it easier not only to read the normal code, but also to reason about the error-handling code and to analyze it [7,15,18].

Proper use of exception handling consists of two separate responsibilities: error detection and error handling. In a stand-alone application, the code that detects an exceptional situation often also directly handles it in the way that is appropriate for the particular application. In a library designed for use by different clients in different contexts, however, the library designer cannot decide on one particular handling but must delegate this decision to the client. For any failing procedure that is visible outside the library, clients should be able to handle an exceptional situation on their own terms. For that, however, often more information is needed than just which kinds of exceptions might be active. If clients want to retry the failing operation, for example, or to ignore the exception, they need to know whether they can safely do so. If the exception has left the program in an inconsistent state – for example, with leaking resources or invalidated data type invariants – it will be incorrect to continue.

The so-called *exception-safety classification* supplies the required information by classifying procedures according to three different safety levels. The classification was introduced during the standardization of the Standard Template Library

* Corresponding author.
E-mail addresses: gustav.munkby@chalmers.se (G. Munkby), schupp@chalmers.se (S. Schupp).

0167-6423/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.06.004

Basic: the invariants of the program are preserved and no resources leak.
Strong: in addition to the basic guarantee, the operation provides rollback semantics in the event of an exception.
No-throw: the operation does not throw an exception.

Fig. 1. Exception-safety guarantees.

(STL) in C++ as part of the contract between library components and users. Although the classification itself is language-independent, it is applied most systematically in C++, for example, during the review process in the Boost organization for library standardization. Classifying the *exception-safety guarantee* of a procedure requires detecting the "worst" possible control flow of the procedure, i.e. the biggest possible change or inconsistency that could occur when an exception leaves the procedure. Since the control flow usually contains a great number of paths that are not directly visible at source-code level, it is in practice difficult to correctly classify a procedure. Passing a parameter, leaving a scope, or calling a sub-routine are all examples where the compiler creates or includes code that complicates the control flow and must be traced even if it ultimately does not raise an exception. Thus, even if one is familiar with the semantics of the source language, it is not trivial to follow the flow of control by hand. Recently, a modified version of Gcc, called EDoc++ [4], has become available that identifies and warns about dangerous exceptional control flow. Yet, EDoc++ does not attempt to deal with the exception-safety guarantees.

On top of the exceptional control flow we have designed and implemented the first automated analysis for exception-safety guarantees. The analysis conservatively approximates the *strong* and the *no-throw* guarantee, the two strongest guarantees. The core idea of the algorithm is to partition the set of all possible control-flow graphs into five equivalence classes, introduce a lattice structure on this set of equivalence classes, and define a data-flow analysis over the annotated abstraction of the control-flow graph. The analysis iteratively derives the equivalence class of an entire control-flow graph from the equivalence classes of its subgraphs; as we will show, two passes over the control-flow graph suffice. The analysis is characterized as an interprocedural, backwards data-flow analysis. Since it obtains the final exception-safety classification by combining the results of subclassifications, the analysis is also compositional.

A prototype of the analysis is implemented in the BangSafe tool set, which interfaces the Elsa parser for C++ and handles a reasonable and relevant subset of C++ programs. BangSafe itself is implemented in Ruby.

The paper is organized as follows. In Section 2, we discuss the terminology and motivation of the exception-safety classification. The analysis is first outlined in Section 3, which describes the overall architecture, and then discussed in detail in Section 4, which presents the equivalence classes, and Section 5, which contains the main algorithm. Section 6 puts together the various parts of the analysis and Section 7 provides examples, including an example of a false positive. The precision of the analysis is discussed in Section 8. A summary of related work (Section 9), an evaluation of the analysis combined with an outline of future work (Section 10), and a final summary (Section 11) conclude the paper.

2. Exception safety

Programming languages like Java and C++ provide so-called exception specifications for annotating a procedure with the types of exceptions that might be raised by a procedure. Exception specifications make it possible to check for certain classes of exceptions whether or not they can escape a procedure. They can be thought as part of the contract between caller and callee, because they make explicit for which exceptions the caller is responsible. Yet, handling exceptions properly requires not only an understanding of what exceptions might be activated, but also knowing what implications a failure has if the program was allowed to continue. The exception-safety classification categorizes procedures into one of three different safety levels, namely the basic, the strong, and the no-throw guarantee [1].

Fig. 1 summarizes the three exception-safety guarantees as they are usually spelled out. The basic guarantee is the weakest, and states that the procedure does not invalidate any of its invariants if an exception is thrown. It ensures that no resources leak and that all data-type invariants remain valid. The strong guarantee, next, additionally specifies that if an exception is thrown from an operation, then the program state shall be the same as before the operation started. The strong guarantee ensures that any detectable changes, even operations such as creating or moving objects, are undone before the exception leaves the context. Finally, the no-throw guarantee means that the procedure may not throw any exceptions. For example, if inserting an element into a container provides the no-throw guarantee the operation will always succeed. If the operation might fail but the original container content is preserved, the strong guarantee is fulfilled. The basic guarantee only ensures that the container is still in some valid state, meaning that any random content will do.

Classifying safety of exception handling is no new idea. Already in the late 1970s, Cristian introduced the notion of procedures that are *weakly* and *strongly tolerant* towards exceptions [5]. In difference to the exception-safety guarantees, Cristian's tolerance levels are explicitly specified on an exception-by-exception basis. If a procedure fulfills the requirements of the strong exception-safety guarantee, Cristian's classification specifies it as weakly tolerant towards all exceptions occurring during its execution. A procedure that is strongly tolerant towards an exception can actually fix the problem that the exception signals, similar to the no-throw exception-safety guarantee.

Exception-safety classification in C++ was introduced during the implementation of the C++ standard library [2], as part of the contractual specification between a procedure and its clients. Nowadays, the exception-safety classification is established and widely used. It is a standard topic in C++ courses [21], used in the standardization process [6], and

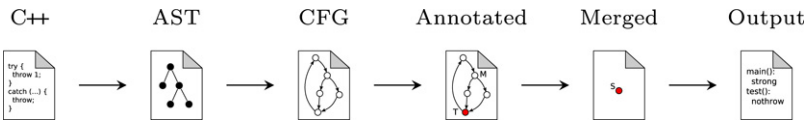


Fig. 2. Transformations of the analysis.

⟨*function*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩ '(' ⟨*argument*⟩* ')' ⟨*compound*⟩

⟨*compound*⟩ ::= '{' ⟨*stmt*⟩* '}'

⟨*stmt*⟩ ::= ⟨*compound*⟩ | ⟨*loop*⟩ | ⟨*branch*⟩ | ⟨*try*⟩ | ⟨*expression*⟩ ';'

⟨*loop*⟩ ::= 'for' '(' ⟨*expression*⟩ ';' ⟨*expression*⟩ ';' ⟨*expression*⟩ ')' ⟨*compound*⟩

⟨*branch*⟩ ::= 'if' '(' ⟨*expression*⟩ ')' ⟨*compound*⟩ 'else' ⟨*compound*⟩

⟨*try*⟩ ::= 'try' ⟨*compound*⟩ ⟨*handler*⟩+

⟨*handler*⟩ ::= 'catch' '(' ⟨*argument*⟩ ')' ⟨*compound*⟩

⟨*expression*⟩ ::= ⟨*literal*⟩ | ⟨*operator*⟩ | ⟨*call*⟩ | ⟨*throw*⟩ | ⟨*assignment*⟩

⟨*call*⟩ ::= ⟨*identifier*⟩ '(' ⟨*expression*⟩* ')'

⟨*throw*⟩ ::= 'throw' ⟨*expression*⟩

⟨*assignment*⟩ ::= ⟨*expression*⟩ '=' ⟨*expression*⟩

Fig. 3. Supported input grammar.

supported by a number of idioms [22], most notably the *Resource-Acquisition-is-Initialization (RAII)* [20] idiom for exception-safe resource management. In our analysis, we therefore target C++. Libraries in other languages can, in principle, benefit from the exception-safety classification as well, but each language introduces its own quirks. In Java, for instance, the virtual machine can inject exceptions asynchronously, which complicates control flow considerably [16]. Our static analysis conservatively classifies procedures as providing the strong or the no-throw guarantee, by detecting the existence of state modifications by a procedure and exiting exceptions. It assumes that the full source code is available, so that all throw statements can be syntactically detected. It furthermore assumes that all state modifications happen through object-level assignments, thus can also be detected syntactically. In Section 8 we discuss how these assumptions affect the precision of the analysis. Even though the analysis might report false positives, we claim that our analysis tool still simplifies the manual exception-safety classification: whenever a program is classified as not exception-safe, the offending paths are reported to the user. By manual inspection of the offending path, a precise classification can quickly be determined.

3. Architecture

Our exception-safety analysis is realized as a series of transformations. Fig. 2 gives an overview of the stages from the initial C++ input to the output containing the exception-safety classification. The first stage parses the input into an abstract syntax tree. In the next stage, a control-flow graph is constructed using the information from the abstract syntax tree. The stages three and four form the main algorithm: the third stage prepares the analysis by annotating the control-flow graph with initial information about exiting exceptions and state modifications; this annotation is based on a syntactic analysis. The fourth, and most important, stage then combines these annotations into one single annotation per procedure to compute the actual classification; the details will be discussed in Sections 4 and 5. The last step, finally, interprets the top-level annotation and emits the classification.

Fig. 3 shows a segment of the grammar of the supported target language. The described language is characterized in two ways:

- It centers around the features of exception handling.
- It contains a subset of executable C++, to ensure that standard parsers are applicable.

Additionally, the target language includes all features that have a simple and straightforward mapping from C++ to the terminals and non-terminals used in Fig. 3. In the interest of readability, we have omitted those features.

The transformations are implemented as a set of command-line applications in Ruby, where each of them allows presenting and visualizing the various stages of the analysis in different ways. The most important tools include BangGraph, which produces an annotated control-flow graph for each procedure, and BangSafe, which runs the main algorithm and produces a trace with the exception-safety classification and invalidating paths.

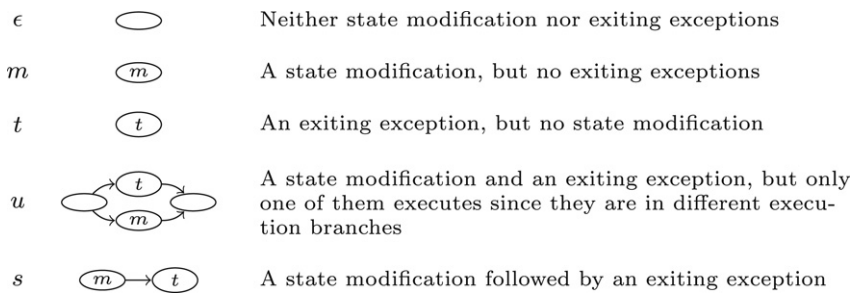


Fig. 4. Abstract representation.

4. Abstract representation

As always in static analysis, the key is to find an abstract representation that is simple enough to solve the problem efficiently, but still ensures that the results are sufficiently precise to interpret the original problem. To check the strong guarantee we must examine whether the program state at exceptional procedure exits is the same as at procedure entry. Therefore, we keep track of state modifications, exiting exceptions, and their ordering.

Our abstract representation is a mapping of a control-flow graph to one of five equivalence classes, abbreviated as m, t, u, s, and ϵ. We denote by m ("modification") and t ("throw") all control-flow graphs that contain only either state modifications (m) or exiting exceptions (t), but not both. Note that an exception is exiting only if it is not caught by a local handler. By u, we denote control-flow graphs that contain m and t subgraphs but embedded in different branches so that no program can execute both, and by s those control-flow graphs where a modification precedes an exiting exception. All other control-flow graphs fall in the class of ϵ graphs. Fig. 4 lists the five equivalence classes, including the smallest representative graph and a description of its characteristics. The short names, m, t, u, s, and ϵ, will be used throughout the entire paper.

Given above abstract representation, there exists a simple mapping between the equivalence classes and the exception-safety guarantee obtained by a procedure. Both checked guarantees can be represented by two different equivalence classes, one with state modification in the normal execution and one without: the classes ϵ and m map to the no-throw guarantee, and the classes t and u map to the strong guarantee. Therefore, the strong guarantee is potentially violated when a graph falls in the equivalence class s and the no-throw guarantee is violated when the graph is labeled t, u, or s.

We define the following partial ordering on the five equivalence classes:

$$\epsilon < \{m, t\} < u < s. \tag{1}$$

This partial ordering captures the intuition of an increasing threat to the strong guarantee and will guide the formulation of the data-flow analysis in the following section.

5. Main algorithm

The main analysis algorithm computes the classification of the control-flow graph of a procedure as one of the five equivalence classes presented in the previous section. In this section we describe the main ideas of annotation and propagation first informally, then we formalize them as a *kill-gen* data-flow analysis [9,10,14] and discuss the properties of the resulting algorithm.

5.1. Annotation and propagation

The analysis starts off by identifying all nodes that correspond to non-local modifications or exceptions that exit the procedure, and annotates them with m and t. This annotation is based on a syntactic analysis of the source-code expressions.

The classifying algorithm itself then considers the syntactically annotated nodes as single-element graphs, classified according to the equivalence classes in Fig. 4, and uses them as the base to incrementally combine more and more annotations, corresponding to larger and larger subgraphs, until the whole procedure is covered. To combine node annotations, we introduce two operations: one for sequencing and another for branching control-flow graphs.

Formally, *sequence* and *branch* are defined as binary operations on the set of equivalence classes $L = \{\epsilon, m, t, u, s\}$, listed in Fig. 4. The definition follows the intuitive understanding of the two operations, and uses the order defined in Eq. (1).

$$\text{branch}(x, y) = \begin{cases} u & \textbf{if } (x = m \wedge y = t) \vee (x = t \wedge y = m) \\ \max(x, y) & \textbf{otherwise} \end{cases} \tag{2}$$

$$\text{sequence}(x, y) = \begin{cases} s & \textbf{if } (x \geq m \wedge y \geq t) \vee (x \geq t \wedge y \geq m) \\ \max(x, y) & \textbf{otherwise}. \end{cases} \tag{3}$$

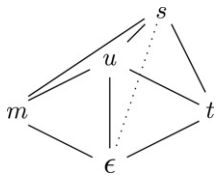


Fig. 5. Lattice.

The *branch* operation represents a choice between the execution of two alternative graphs. This operator, as expected, originates from the branch statement in the grammar (see Fig. 3). Since the exception-safety guarantee is a worst-case guarantee, our analysis is interested in possible violations of the guarantee. In most cases we can therefore discard one of the branches and just keep the worse. There is one exception, though, where both branches are equally "bad", namely when one branch leads to m and the other to t. In this case, the resulting graph is classified as u.

The *sequence* operation corresponds to the sequential execution of two graphs, and is used for the remaining rules (see Fig. 3). For *sequence* we return s if there is at least a state modification in one of the inputs and an exiting exception in the other. Our abstraction makes no difference between modifications; thus we note that a series of consecutive m can be replaced with a single m, resulting in pessimistic treatment of undoes. One might wonder why a t annotation, which represents an exiting exception, can be sequenced with another graph. In C++, the execution of automatic destructors means an exiting exception is no longer necessarily the last expression. Note, however, that a catch block is not executed after a t annotation since a caught exception is not exiting.

5.2. Data-flow analysis

We express the analysis in terms of the general approach for data-flow analysis [9,10,14]. In the general approach, information contained in a *lattice* is propagated through the control-flow graph using *flow functions*. In our problem, the set of equivalence classes, L, together with the *branch* operation form the lattice, and the *sequence* function defines the flow functions. The general approach enables a standard iterative strategy, given that the flow functions are *monotonic*. If the lattice is also *distributive*, the iterative solution obtained is the *meet over all paths* solution, which is the optimal result for any data-flow analysis.

We define a lattice over L from the abstract representation together with the *branch* operation as the lattice join operation. The definition of the corresponding meet operation works in the opposite way, and returns the least input, or ϵ if there is no least input. We present the resulting idempotent and distributive lattice in Fig. 5; the lattice properties are easy to verify.

Following standard notation from data-flow analysis, we now define the two data-flow equations that should hold for every node n of the control-flow graph of a procedure p: the *in* annotation, which captures the intermediate classification of the control-flow graph reachable from n but excluding the node itself, and the *out* annotation, which also includes the initial annotation, $gen[n]$, of the node n:

$$in[n] = \begin{cases} \epsilon & \textbf{if } n \in exits_p \\ \text{branch}\{out[s] | s \in succ(n)\} & \textbf{otherwise} \end{cases}$$

$$out[n] = \text{sequence}(gen[n], in[n]).$$

The exit annotation is in the normal case just the *branch* operation applied to the entry annotation of all successor nodes. If the node is an exit node, there are no successors and we initialize the exit annotation to ϵ. The *out* annotation is simply the *in* annotation sequenced with the initial syntactic annotation, $gen[n]$, of the node.

Sequencing the initial annotation with the *in* annotation represents the flow function. From the formal definition of *sequence* in Eq. (3), we can immediately conclude that the flow function is monotonic, i.e.,

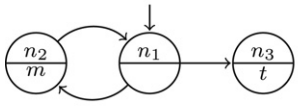
$$\forall a, x \in L: x \leq \text{sequence}(a, x). \tag{4}$$

For the case when *sequence* returns s, the inequality is trivially true, since s is the largest value. For the other case, the inequality is also trivially true, since $x \leq \max(a, x)$.

Monotonic flow functions allow us to express the analysis in terms of an iterative algorithm where we start by assuming that the *out* annotation for every node is ϵ and then use the data-flow equations as functions to iterate through the control-flow graph nodes updating all *in* and *out* annotations until a fix-point is reached. When a fix-point is reached, the *out* annotation for procedure p's entry, $out[entry_p]$, contains an annotation that reflects the control-flow graph of procedure p. When later analyzing a procedure that calls p, we can attach this annotation to the call-site without further computation. The single annotation for a whole procedure means that the algorithm scales well and that it is compositional.

5.3. Correctness

We now show the partial and total correctness of the analysis. Termination follows directly from the fact that we use iterative data-flow analysis, where our flow function is monotone and our lattice has finite size. For our analysis, however,



	in_1	out_1	in_2	out_2
n_3	ϵ	t	ϵ	t
n_2	ϵ	m	u	s
n_1	u	u	s	s

Fig. 6. Control flow requiring two passes.

we can more precisely claim that two iterations will suffice. We divide the argumentation into two cases. First we assume there are no loops, and show that a single iteration is sufficient. We then introduce loops and show that two traversals suffice.

Our analysis is categorized as a backwards data-flow analysis, since information is carried from successor to predecessor. In a backwards data-flow analysis, the nodes of the control-flow graph are updated in a postorder traversal, where successors are visited before predecessors. Assuming no loops, the postorder traversal defines a topological ordering, thus ensuring that the equations for succeeding nodes are already solved before the current node is visited. The correctness of the resulting annotation for the current node follows directly from the correctness of the definitions of the combining functions (Eq. (2), Eq. (3)).

By introducing loops, a topological ordering of the control-flow graph nodes is no longer possible. Fig. 6 contains the smallest possible example graph that will not be correctly classified after a single iteration of the analysis. The correct annotation for the node n_1 is s. After one iteration, however, the analysis produces only u. When node n_2 is analyzed, the *out* annotation of the node n_1 has not yet been computed, and therefore still carries the initial value ϵ.

For any loop-entry, we note that the reachability of other nodes is not affected by the existence of back-edges. As it is easy to see from Fig. 4, the only classification not solely determined by reachability is s. Since both a state modification and an exiting exception must be reachable for a procedure to be annotated with s (see Eq. (3)), the annotation for the first iteration must be at least u. Applying another iteration, the loop-entry will be annotated s if and only if the loop contains an m; from there, its annotation cannot change any further. We therefore conclude that two iterations of the above traversal of the graph ensure that the procedure-entry is annotated with the correct classification of the whole procedure, with respect to Fig. 4.

5.4. Complexity

The complexity of the algorithm is dominated by the costs for graph traversal. Given that we can perform $O(1)$ table lookup, the complexity of the whole analysis is, $O(|V| + |A|)$, where $|V|$ denotes the number of nodes and $|A|$ the number of edges.

6. Putting it together

We can now return to Fig. 2 in Section 3 and fill in some technical details. The first step is to parse C++, which is performed by the third-party program Elsa, developed by McPeak et al. [11,12]. Elsa constructs an abstract syntax tree, annotated with type-checking information. It claims to support most of standard C++, but we have not verified that claim. Elsa emits the abstract syntax tree as an Xml-hierarchy. From there, the BangSafe toolset picks up the program representation and maps it to its own internal representation in Ruby.

The second step lowers the abstract syntax tree to an expression-level control-flow graph with support for interprocedural exception-flow. The control-flow graph is at expression level because the C++ language constructs for exceptions and state modifications are all expressions. To be able to construct the interprocedural exception-flow correctly, we must determine the potential exceptions raised by called functions, including recursive and dynamically dispatched functions, or calls through function pointers. In our prototype we simply conservatively approximate unknown function calls by s annotations. Furthermore, the control-flow graph is imprecise insofar it assumes that all syntactically possible branches are executed and that all loops terminate. The actual analysis happens during the third and fourth transformation, described in detail in Section 5. By the end of the fourth transformation, the root node of the control-flow graph contains the final classification of the procedure: it satisfies the no-throw guarantee when labeled ϵ or m, satisfies the strong guarantee when labeled t or u, and potentially violates the strong guarantee when labeled s.

The last step organizes the command-line output of the analysis. If the previous step has determined that the strong guarantee might be invalidated, the tool emits not just the diagnostics but also the paths possibly leading to invalidation. This is done in the simplest of ways by just enumerating all possible paths through a procedure; to ensure termination, loops are contracted in a preprocessing step. Because of the enumeration, the complexity of the last step is exponential in the number of sequential branches where the strong guarantee has been violated.

7. Examples

In this section, we walk the reader through three different examples, to illustrate how the different transformations of our algorithm work. The first and the second example demonstrate correctly detected guarantees, and the last example

```
1 template<class T>
2 vector<T>::vector(size_type n, const T & val) {
3   v = cpp_malloc<T>(n);
4   space = last = v +  n;
5   for (T* p=v; p != last; ++p) {
6     new (p) T(val);
7   }
8 }
```

Fig. 7. Source code for naive approach.

demonstrates a false positive. All examples come from Stroustrup [21], but are slightly simplified. Most notably, we replaced class-based allocation by two functions. We also replaced the templates in Stroustrup's examples by instantiated templates, since our tool does not support uninstantiated templates.

All three examples use implementations of a constructor for a *vector* class in the style of the C++ standard library, which is implemented using a dynamically allocated buffer and three pointers. The constructor we are looking at allocates space for n elements and constructs as many copies of an initial value in the allocated space.

The problems discussed in Stroustrup's examples come from the fact that copy construction of objects can fail. To enforce failure, for demonstration purposes, we devise a *bomb* class, which is specifically designed to throw exceptions when being copy-constructed.

7.1. The naive approach

The first example is listed in Fig. 7. Stroustrup calls it the naive approach. There are two sources of exceptions within the procedure:

- *cpp_malloc* throws if no memory is available (line 3).
- Placement *new* uses the copy constructor of the element type T to copy the initial value *val* (line 6). This copy constructor might throw.

By careful manual analysis, it is possible to figure out that if the latter throws, one has already allocated memory, which would be necessary to free. Yet, memory leakage is not the only problem. One might have actually successfully constructed a few copies before a copy construction fails. A correct program also needs to destruct these objects.

Our analysis will produce a trace describing that the strong guarantee is invalidated because it is possible to perform a *cpp_malloc* that is followed by at least one copy construction before throwing an exiting exception.

The first annotation pass generates the control-flow graph in Fig. 8, which has abstracted from the original control-flow graph all irrelevant nodes and kept only the ones with information about exception raises and state modifications; for technical reasons BangGraph introduces an exit node for each original node in the control-flow graph and keeps all annotations in these exit nodes. As we can see from the graph, at this point all inner nodes are labeled with m or t (see Fig. 4) and the root node, i.e., the procedure entry, with e (representing the ϵ in Fig. 4). After the next step, some nodes will change to u or s, and the root node will contain the final classification of the procedure.

A closer look at the BangGraph output in Fig. 8 reveals that the analysis is overly conservative: we have five different m annotations, but only the one for the *cpp_malloc* invocation (line 3) and the one for the copy construction in the new-expression (line 6) are needed. The other assignments are updates to instance variables of the constructor, which will not propagate outside the failing constructor and thus ideally should not produce m annotations. However, support for the special semantics of constructors has not yet been implemented.

If we ignore the back-edge in Fig. 8 and consider all successful copy constructions as a single operation, we can identify five distinct execution paths through the procedure:

- Failing early with *cpp_malloc*.
- Succeeding by skipping the loop, avoiding both normal and exceptional exits of the new-expression.
- Succeeding by completing the copy construction of all entries, skipping the exceptional exit of the new-expression.
- Failing at the first copy construction, going directly to the exceptional exit of the new-expression.
- Failing at a subsequent copy construction, entering first the normal exit of the new-expression, continuing to the exceptional exit.

Running BangSafe on the example, produces the trace in Fig. 9. This trace has not been pruned, to show the volume of information within the logfile. The interesting portion of the output starts at the line containing the constructor *vector<bomb>::vector<bomb>*, which shows that the procedure is correctly annotated with s and lists an invalidating path. This path describes the worst case where both *cpp_malloc* and one successful copy construction have been applied before an exception exits the constructor. The copy construction is, as explained earlier, embedded in the new-expression.

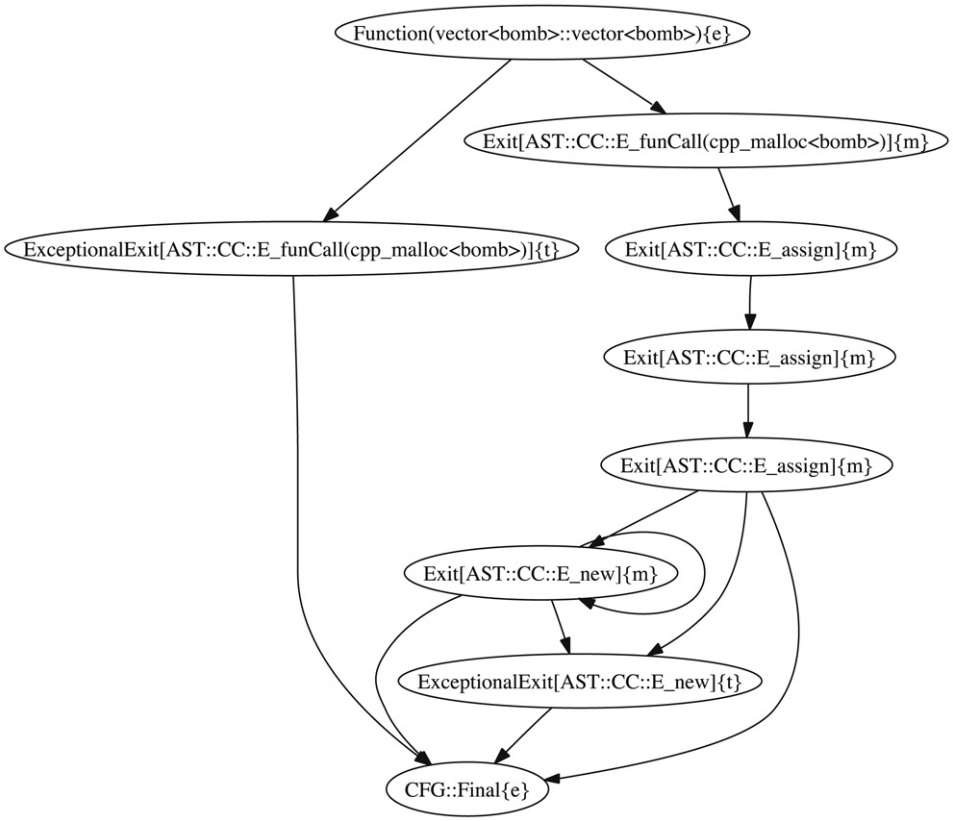


Fig. 8. BangGraph for naive approach.

7.2. The naive approach, take two

Now suppose that we instantiate the same constructor with *int* instead of *bomb*. Then the example provides the strong guarantee because the copy construction of *int* cannot throw. The analysis can confirm that. We do not list the trace separately, but the result is part of Fig. 9, where the entry for *vector<int>::vector<int>* is *u*, which indicates that no state modification took place before the exception was thrown; thus the strong guarantee holds. This result is correct, since only allocation can fail, but all allocation takes place before any state modification.

The output of BangGraph for the instantiation with *int* looks identical to Fig. 8, except that the node labeled as an exceptional exit from the new-expression together with its attached edges no longer exist.

7.3. The revised example

As a final example, we discuss a case where our analysis produces a false positive. In this example, shown in Fig. 10, Stroustrup solves the exception-safety violation from the naive approach in a straightforward way, by using the standard routine *uninitialized_fill* to construct the objects and wrapping the whole block of code in a try-catch block to ensure that any allocated memory is freed on failure.

As the BangSafe trace in Fig. 11 shows, the analysis has identified a path through the procedure that invalidates the strong guarantee. The path represents successful allocation, *cpp_malloc*, followed by a failing *uninitialized_fill*. Since the result of the allocation is assigned to a variable (line 4) and *uninitialized_fill* causes an exception that exits the procedure, the annotated control-flow graph contains a sequence of *m* followed by *t* for this path. Therefore, the analysis produces an *s* annotation for the whole procedure. In reality this path constitutes no problem—it represents a false positive: the state is restored because the memory allocated is freed (line 9), *uninitialized_fill* actually provides the strong guarantee, and the assignment is to an instance variable in the constructor, thus could never have modified the program state.

The latter source of a false positive, conservative treatment of assignments, can be dealt with relatively easily, as in the naive example, by adding special support for constructor semantics. A more fundamental problem is that the algorithm does not realize that the state changed by *cpp_malloc* is reverted by *cpp_free*. Using our syntactic abstraction, this restoration cannot be detected. Not being able to detect reversals is also the reason why *uninitialized_fill* is incorrectly classified with *s* instead of *u*.

```
> bangsafe vector-naive.dump
Annotation for 'operator='ise
Annotation for '~bad_alloc'ise
Annotation for 'operator='ise
Annotation for '~vector'ise
Annotation for 'bad_alloc'ise
Annotation for 'operatornew'isu
Annotation for 'cpp_malloc<>'isu
Annotation for 'cpp_malloc<bomb>'isu
Annotation for 'operatordelete'ise
Annotation for 'bomb'ise
Annotation for 'bomb'isu
Annotation for 'operator='ise
Annotation for 'operator='ise
Annotation for '~bomb'ise
Annotation for 'cpp_malloc<int>'isu
Annotation for 'cpp_free<>'ise
Annotation for 'vector<bomb>::vector<bomb>'is
  Detected invalidating path:
  > m:3:AST::CC::E_funCall(cpp_malloc<bomb>)
  > m:3:AST::CC::E_assign
  > m:4:AST::CC::E_assign
  > m:4:AST::CC::E_assign
  > m:6:AST::CC::E_new
  > t:6:AST::CC::E_new(AST::CC::E_throw)
Annotation for 'bad_alloc'ise
Annotation for 'vector'ise
Annotation for 'vector<int>::vector<int>'isu
Annotation for 'vector'ise
Annotation for 'operatornew'ise
Annotation for '~vector'ise
Annotation for 'vector<T>::vector<>'isu
```

Fig. 9. BangSafe for naive approach.

```
1  template <class T>
2  vector<T>::vector(size_type n, const T& val)
3  {
4          v = cpp_malloc<T>(n);
5          try {
6                  uninitialized_fill(v, v+n, val);
7                  space = last = v+n;
8          }
9          catch (...) {
10                 cpp_free(v);
11                 throw;
12         }
13 }
```

Fig. 10. Revised source code.

```
> bangsafe vector-revised.dump
...
Annotation for 'vector<bomb>::vector<bomb>' is s
  Detected invalidating path:
  > m:4:E_funCall(cpp_malloc<bomb>)
  > m:4:E_assign
  > m:6:E_funCall(uninitialized_fill<bomb*,bomb>)
  > m:10:E_funCall(cpp_free<bomb>)
  > t:11:E_throw()
...
```

Fig. 11. BangSafe for revised example.

8. Precision

Almost all static analyses are imprecise insofar they might produce false positives. Very practically, thus, their usefulness depends on the actual number of false positives and their source. We have not yet performed extensive benchmarking, but we can characterize the sources of imprecision of the presented analysis and discuss possible countermeasures.

The following four abstractions lessen the precision of the algorithm:

(1) Treating every state modification as irreversible. This essentially prevents the algorithm from detecting cases where the state is first invalidated, and then revalidated by a later state modification—a pattern very common in real code.
(2) Detecting whether a variable is only local in scope, is only done syntactically. Modifications to references and dereferencing pointer variables will therefore always lead to *m* annotations. This loss of precision happens early, in the initial annotation of the control-flow graph, and is propagated further by the iterative data-flow analysis. In the interprocedural case we lose further precision since we do not map modifications against passed arguments.
(3) Merging several branches and always taking the worst. This will ensure that no possible problems are missed, but will in some cases result in a pessimistic judgment of a procedure.
(4) Pessimizing on constructs with nondeterministic execution order. To ensure that all possible orders of execution are included without making the graph too big, procedure call arguments and other nondeterministic ordering constructs are converted to loops. Unfortunately, this introduces infeasible execution paths.

Of those four abstractions, the first two appear to have the biggest impact on the overall precision and deserve further investigation. In the first case, we believe that the user could provide annotations for a modification and its reversal. In particular, classes that implement the *Resource-Acquisition-is-Initialization (RAII)* idiom could be annotated as such, which means that the analysis could safely ignore them. In the second case, one could exploit the semantics of C++ to determine which state modifications result in local effects only and will not escape the procedure boundaries. Additionally, one could consider integrating a points-to analysis. Such points-to analysis would be most effective in combination with an improved interprocedural analysis, since the analysis could then bind modifications of arguments to local variables in the caller instead of treating them as global state modifications.

Another source of imprecision in the prototype implementation is the pessimistic algorithm to determine exceptional control flow, discussed in Section 6. Naturally, any imprecision in the exceptional control flow results in imprecision in the analysis output.

9. Related work

Our work applies techniques from data-flow analysis to detect possible correctness problems caused by uncaught exceptions. It therefore draws on previous work on exception safety, but also benefits from techniques in data-flow analysis that are established and well understood.

9.1. Exception safety

Despite of the practical relevance of exception safety, only little theoretical work or practical support exist. Aside from EDoc++ [4], the only automation available until recently was based on a test suite for the C++ standard library, which Abrahams developed as part of the implementation of STLport [1]. The testing technique has since been generalized and incorporated into the Boost *Test* library [17]. Testing requires tailoring the test-suite to the library or program being tested. It is on the one hand more powerful than our approach insofar it enables checking even the basic guarantee. On the other hand, it must be adapted for every target.

Alexandrescu et al. describe a manual analysis for exception-safety classification [3]. The abstraction in their approach is similar to ours, but based on a classification of procedures, not of control-flow graphs: they distinguish between *pure* procedures, which neither modify program state nor cause any side effects, and *impure* procedures, which are not pure. We can map our five equivalence classes in a straightforward manner to their terminology: s, u, and m each corresponds to an impure procedure, t and ϵ to a pure procedure. While we capture the final level of exception safety directly in one of the equivalence classes, Alexandrescu et al. need to further distinguish which kind of guarantee an impure procedure provides; pure procedures give the strong guarantee by definition. Like our analysis, theirs can check the strong and no-throw guarantees, but not the basic guarantee. The most important difference, yet, is that their algorithm is designed for manual application, thus pays less attention to the details necessary for automation.

9.2. Static analysis

In sharp contrast to the area of exception safety, there is a long history (see, e.g., [9,10]) and a large body of work on data-flow analyses and static analyses techniques in general—too much to list or discuss comprehensively. Static analyses are available across all higher programming languages and in almost all phases of the software development cycle. Applications range from program validation over optimizations to re-factoring, either embedded in a tool chain or organized as stand-alone programs. In the latter case, they typically flag source-code artifacts that are likely to be bugs; much as the exception-safety analysis does.

Static analysis is an umbrella term that encompasses not one generic technique but rather a variety of highly specialized techniques, which mostly deal with one particular language construct—program variables, references, and loops, for example, but also virtual functions or array bounds. Not many analyses, however, address exception-handling mechanisms at all. In our analysis, where we rely on interprocedural exception-flow, we were inspired by similar analyses by Robillard et al. for Java [16], and Schaefer et al. for Ada [19].

The data-flow analysis itself that forms the core of the exception-safety analysis is very similar to the classical analyses in optimizing compilers [14]; we essentially adopt the standard "ingredients" of flow graph, equations, and iterative solving to the representation and particularities of exception safety. Insofar state modification are propagated, the analysis can be considered an example of the well-known family of gen/kill analyses.

10. Evaluation and future work

The current prototype implementation can deal with programs in a subset of C++ that already allows us to test the design of the analysis-algorithm and to put on an experimentally validated basis the ideas that underlie our static analysis of the strong and no-throw exception-safety guarantee.

Ultimately, we want to be able to provide a fully-fledged tool for exception-safety classification. To that end, we would like to investigate how to check the basic exception-safety guarantee. Generally, statically checking the basic guarantee is similar to proving program correctness, in the sense that it requires a complete understanding the program invariants. It would be interesting to see whether our analysis could be extended to handle a smaller portion of the basic guarantee, such as the absence of resource leaks.

In a first step, however, we have to evaluate the abstractions of the analysis and its precision by performing real-world benchmarking. Testing with real-world libraries presupposes that the analysis can deal with more features of the C++ language than it currently does. Most of these features we expect to impose no difficulties to the algorithmic logic of the analysis, although they might be technically non-trivial to implement. The thesis of the first author goes into detail about several possible extensions [13]. Of all possible extensions, the most interesting, and also the most complicated one, is the support for uninstantiated templates, which will require modifications of the analysis.

Enabling the analysis to support uninstantiated templates is not easy, mainly because one must identify all implicit subroutine calls that affect the final exception-safety classification. In C++, many expressions can implicitly lead to procedure calls, including automatic type conversions, overloaded operators, and return statements. If a template is instantiated, we can rely on the parser to lower the abstract syntax tree to explicitly include any procedure calls as part of its type checking. For uninstantiated templates, however, only limited type checking can take place.

Given the relevance of uninstantiated templates, one might wonder how useful a prototype is that provides support for instantiated templates only. Yet, we claim that a focus on instantiated templates is a proper simplification for a first prototype, particularly since we believe that the current analysis can be embedded in the more developed one that accomplishes full template support.

One might also wonder whether the upcoming introduction of *concepts* in the next major version of C++ will make support for templates easier [8]. In short, concepts allow expressing constraints on template parameters and therefore enable the compiler, without specializing a template, to determine the set of candidate functions that could be called from a specific call-site. Concepts can thus restrict the behavior of all admissible templates, but they cannot ensure that all candidates behave identically in terms of our analysis: we need to know what types of exceptions are thrown and whether state-modifications can take place inside a candidate, but those properties are not considered during the concept check. It is conceivable, however, that one can specify these properties of interest in terms of concepts and then explicitly state the mapping from a procedure to the provided exception-safety guarantees. Our analysis could then be adopted to produce such mappings automatically.

Another practical question is the scalability of the analysis. The complete examples, including template instantiations, the vector class, and needed supporting procedures, consist of less than 100 lines of C++ source code each. By design, however, the analysis is prepared to handle larger programs. Its costs are linear in the size of the control-flow graph, it is an interprocedural analysis and it is compositional: since all required information is contained within the equivalence-class annotations and the final result is kept in the root node of the control-flow graph of a program, an already analyzed program that becomes a subprogram of a larger program can be represented just by its root node, annotated with the classification. In its current incarnation, the analysis is a whole-program source-code analysis, but refining it to a fragment analysis seems possible.

11. Summary

An important part of the contract between a library component and its client is the exception-safety guarantee that the library component assures; the actual safety level defines the options the client has when handling an exception. In particular in C++, exception-safety guarantees are part of the review process of libraries and, conversely, many library components are designed with the strong exception-safety guarantee in mind. Until now, however, exception-safety guarantees have to be determined by hand.

We have designed and implemented a static analysis that conservatively determines whether a library procedure fulfills the requirements of either the strong or the no-throw exception-safety guarantee. The analysis also allows a library developer to test whether the exceptional behavior of the library procedures is as intended.

It is not always necessary to demand the strong guarantee; depending on the application and its fault tolerance level, but also on the design of a component, its degree of mutability and its number of valid states, it might suffice to ask for the basic guarantee. Even though the analysis is designed to support only the two stronger guarantees, we believe that it is helpful even for the basic guarantee. The analysis explicitly lists the cases where the strong guarantee might be invalidated and thereby highlights the locations where the basic guarantee could be invalidated.

In summary, the core idea of the underlying algorithm is to introduce five equivalence classes of control-flow graphs and define a lattice structure over them. Based on an initial (syntactic) annotation of the control-flow graph with state modifications and exiting exceptions, a backwards iterative data-flow analysis propagates this annotation in an efficient and compositional manner.

Acknowledgments

This work was in part supported by the project CEDES, which is funded within the Intelligent Vehicle Safety Systems (IVSS) program, a joint research program by the Swedish industry and Government. We thank the reviewers for their insightful comments and suggestions.

References

[1] D. Abrahams, Exception-safety in generic components, in: M. Jazayeri, R. Loos, D.R. Musser (Eds.), Generic Programming, in: Lecture Notes in Computer Science, vol. 1766, Springer, 1998, pp. 69–79.
[2] D. Abrahams, G. Colvin, Making the C++ standard library exception safe, Tech. Rep. N1086 =97-0048R1, C++ Standards Committee (1997).
[3] A. Alexandrescu, D.B. Held, Smart pointers reloaded (ii): Exception safety analysis, C/C++ Users Journal 21 (12) (2003) 40–44.
[4] B. Costa, EDoc++ Manual. http://edoc.sf.net/EDocManual, Aug. 2007.
[5] F. Cristian, A recovery mechanism for modular software, in: Proc. ICSE '79, IEEE Press, Piscataway, NJ, USA, 1979, pp. 42–50.
[6] B. Dawes, Boost library requirements and guidelines. http://www.boost.org/development/requirements.html, April 2008.
[7] J.B. Goodenough, Structured exception handling, in: Proc. POPL '75, ACM Press, New York, NY, USA, 1975, pp. 204–224.
[8] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G.D. Reis, A. Lumsdaine, Concepts: Linguistic support for generic programming in C++, SIGPLAN Not. 41 (10) (2006) 291–310.
[9] J.B. Kam, J.D. Ullman, Global data flow analysis and iterative algorithms, Journal of ACM 23 (1) (1976) 158–171.
[10] G.A. Kildall, A unified approach to global program optimization, in: Proc. POPL'73, ACM Press, New York, NY, USA, 1973, pp. 194–206.
[11] S. McPeak, Elkhound and Elsa. http://www.cs.berkeley.edu/~smcpeak/elkhound/, August 2005.
[12] S. McPeak, G.C. Necula, Elkhound: A fast, practical GLR parser generator, in: CC'04, in: Lecture Notes in Computer Science, vol. 2985, Springer, 2004, pp. 73–88.
[13] G. Munkby, Design and implementation of an algorithm for the strong exception-safety guarantee in C++, Master's Thesis, Chalmers University of Technology, May 2006.
[14] F. Nielsen, H.R. Nielsen, C. Hankin, Principles of Program Analysis, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.
[15] B. Randell, System structure for software fault tolerance, in: Proc. Internat. Conf. on Reliable Software, 1975, pp. 437–449.
[16] M.P. Robillard, G.C. Murphy, Static analysis to support the evolution of exception structure in object-oriented systems, ACM Transactions on Software Engineering Methodology 12 (2) (2003) 191–221.
[17] G. Rozental, Boost test library homepage. http://www.boost.org/libs/test, Feb. 2007.
[18] B.G. Ryder, M.L. Soffa, Influences on the design of exception handling. ACM SIGSOFT project on the impact of software engineering research on programming language design, SIGSOFT Software Engineering Notes 28 (4) (2003) 29–35.
[19] C.F. Schaefer, G.N. Bundy, Static analysis of exception handling in Ada, Software—Practice & Experience 23 (10) (1993) 1157–1174.
[20] B. Stroustrup, The Design and Evolution of C++, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.
[21] B. Stroustrup, The C++ Programming Language, special ed., Addison-Wesley Professional, 2000.
[22] H. Sutter, Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems and Solutions, in: AW C++ in Depth Series, Addison Wesley, 2004.